PHP. Профессиональный уровень

Урок 7

Тестирование. PHPUnit

Введение в принципы тестирования, TDD, BDD. Знакомство с PHPUnit, написание тестов.

Язык PHP считается легким для изучения. С точки зрения карьерного роста это выражается в том, что разработчик быстро выходит за рамки несложных задач (вроде создания сайтов-визиток или элементарных приложений) и начинает командно работать со сложной бизнес-логикой.

Программист-профессионал задумается: «А если я переделаю работу метода А и добавлю класс Б, не упадет ли все?». После изменения номер N что-то гарантированно перестанет работать так, как надо. Локализация и устранение ошибки может отнять много времени. А юнит-тесты могут свести этот процесс к считанным минутам.

Не только код может осложнить ситуацию. Что, если вы решите сменить хостинг, а на новом сервере будет стоять система не x86, а x64?

На все эти вопросы поможет ответить модульное (или юнит-) тестирование на базе популярного фреймворка PHPUnit.

Что такое модульные тесты?

Модульные тесты предназначены для тестирования отдельных фрагментов кода: методов или функций. Часто их используют для проверки групп методов. Например, в методе **getUserInfo()** ожидаем имя пользователя из метода **getUserName()** и аватарку из **getUserpic()**. **GetUserInfo()** должен вернуть массив. Тогда проверяем, что метод **getUserInfo()** возвращает именно массив с двумя переменными, значения которых мы знаем заранее.

Другие типы тестов

Функциональное тестирование

С помощью функционального тестирования проверяют действие сразу. Например, нужно протестировать открытие страницы «О сайте». Вы заранее знаете, какой там должен быть контент. Скачиваете эту страницу и проверяете ее на соответствие эталону.

Или надо проверить смену пароля: сначала этот пароль изменяем с помощью http-запросов (через php), а потом пытаемся авторизоваться с новыми данными.

Регрессионные тесты

Исправляя ошибку, выполняем список действий, приводящих к этому багу, и проверяем, возникает ли эта ошибка. Например, при изменении пароля он, на самом деле, не менялся. Проверяем факт смены пароля в БД и пытаемся авторизоваться.

Smoke-тестирование

Поверхностное тестирование функционала. Выполняем обычные действия пользователя, и смотрим, возникают ли ошибки вообще.

Front-end тестирование

Клиентский код (например, **Javascript**) проверить с помощью **PHP** проблематично. Кроме того, поведение пользователя может сильно отличаться от того, как вы представляли себе его, когда писали сайт. Можно настроить специальное ПО, которое имитирует действия пользователя: кликает на кнопки, скроллит экран, печатает текст в полях.

Применение PHPUnit

Инструкции по установке расписаны на официальном сайте фреймворка: https://phpunit.de/getting-started.html. Уделим максимум внимания применению **PHPUnit**. Рассмотрим простой пример.

Базовый тест

Предположим, что у нас есть класс **MathClass**, в методе которого производится вычисление факториала от числа (вспоминаем «PHP. Level 1»).

```
<?php
class MathClass{
   public function factorial($n){
      if ($n == 0)
        return 1;
      else
        return $n * $this->factorial($n - 1);
    }
   }
}
?>
```

Проверим корректность работы метода. Пока не будем рассматривать весь класс.

Для создания теста напишем такой код:

```
<?php
require_once 'MathClass.php';
class MathClassTest extends PHPUnit_Framework_TestCase {
    public function testFactorial()
    {
        $my = new MathClass();
        $this->assertEquals(6, $my->factorial(3));
 }
}
```

Код модульного теста строится по следующим правилам:

- Название класса в тесте формируется из названия тестируемого класса плюс «**Test**»;
- Класс для тестирования обычно наследуется от **PHPUnit_Framework_TestCase**;
- Любой тест имеет область видимости **public**, а его название начинается с префикса «**test**»;

- Внутри теста мы применяем один из assert-методов, чтобы выясненить, соответствует ли результат обработки ожидаемому;
- Тесты выполняются в алфавитном порядке: **test_a**, **test_b**, **test_c**. Если нужно выполнить несколько тестов в определенном порядке, надо задать им названия по типу **test_001_***, **test_002** и т. д.

При повторной работе с кодом видно, что для тестирования нужно создать экземпляр класса (метод ведь не статический). После этого вызываем нужный метод, заранее определяя ожидаемые значения. Метод **assertEquals** сравнит первое значение (ожидаемое) со вторым (результат работы метода, актуальное значение) и выдаст результат. Выглядеть при тесте это будет примерно так:

```
$ phpunit MathClassTest
.
Time: 0 seconds
OK (1 test, 1 assertion)
```

Результат выполнения теста «**OK**». Это говорит о том, что все тесты и все assert-ы в них выполнились успешно, т.е. ожидаемые результаты полностью совпали с актуальными. Это говорит о том, что покрытый тестами функционал работает корректно и готов к релизу.

И тут можно бы остановиться на достигнутом, но тогда **PHPUnit** не был бы фреймворком.

Поработаем и с другими его возможностями.

Важно: если вы работаете с версией **PHPUnit** ниже, чем 3.6, в начале теста обязательно нужно подключать библиотеку фреймворка:

```
require_once 'PHPUnit/Framework.php';
```

Провайдеры данных

Зачастую для проверки тесту нужно передать не один набор данных, а целый перечень. **PHPUnit** умеет делать и так, предоставляя нам провайдеры данных – public-методы, которые возвращают массив наборов данных для каждой итерации теста. Чтобы провайдер данных применился, необходимо указать его в теге **@dataProvider** к тесту.

```php
<?php
require_once 'MathClass.php';
class MathClassTest extends PHPUnit_Framework_TestCase {
    /**
    * @dataProvider providerFactorial
    */
    public function testFactorial($a, $b)
    {
        $my = new MathClass();
        $this->assertEquals($b, $my->factorial($a));
    }
    public function providerFactorial()
    {
        return array (
            array (0, 1),
            array (2, 2),
            array (5, 120)
        );
    }
}
```

В результате теста получаем:

```
...
Time: 437 ms, Memory: 7.75Mb
OK (3 tests, 3 assertions)
```

Три точки в начале скрипта – не опечатка. Каждая из них означает один успешно пройденный тест. Если на каком-либо тесте произойдет ошибка сравнения, вместо точки появится буква **F**.

Типы сравнений

Помимо **assertEquals** есть еще много типов сравнений, которые могут применяться в различных ситуациях.

Два самых простых – это **assertFalse()** и **assertTrue()**. Эти методы отвечают за проверку на соответствие полученного значения **false** и **true** соответственно. Далее – уже знакомый нам метод **assertEquals()** и обратный ему **assertNotEquals()**. Используя их, нужно помнить о следующих особенностях:

- при сравнении чисел с плавающей точкой есть возможность указать точность сравнения;
- эти методы используются для сравнения экземпляров класса **DOMDocument**, массивов и любых объектов (в последнем случае равенство будет установлено, если атрибуты объектов содержат одинаковые значения).

Также следует упомянуть **assertNull()** и **assertNotNull()**, которые проверяют соответствие параметра типу данных **NULL**. Этим возможные сравнения не ограничиваются. Полный список можно найти здесь.

Принадлежности

Уже знакомый нам по примерам выше класс проводит несколько тестов, в каждом из которых создается экземпляр тестируемого класса. А это не самое лучшее расходование машинного времени. **PHPUnit** предоставляет механизм принадлежностей теста (**fixtures**). Они устанавливаются

защищенным методом **setUp()**, который вызывается один раз перед началом каждого теста. После окончания теста вызывается метод **tearDown()**, в котором мы можем провести «сборку мусора».

```
class MathClassTest extends PHPUnit_Framework_TestCase {
    protected $fixture;
    protected function setUp()
    {
        $this->fixture = new MathClass();
    }
    protected function tearDown()
    {
        $this->fixture = NULL;
    }
    // ...
}
```

Наборы тестов

Довольно быстро у вас появится много тестов для разных классов. И запускать каждый из них вручную неудобно – лучше применять наборы тестов. Объединяем несколько связанных единой задачей тестов в набор и запускаем его.

Наборы реализованы классом **PHPUnit_Framework_TestSuite**. Вам нужно лишь создать экземпляр этого класса и добавить в него необходимые тесты с помощью метода **addTestSuite()**. Также с помощью метода **addTest()** возможно добавление другого набора.

```php
<?php
require_once 'MyClassTest.php';
class MySuite extends PHPUnit_Framework_TestSuite {
    protected $sharedFixture;
    public static function suite()
    {
        $suite = new MySuite('MyTestSuite');
        $suite->addTestSuite('MathClassTest');
        return $suite;
    }
    protected function setUp()
    {
        $this->sharedFixture = new MyClass();
    }
    protected function tearDown()
    {
        $this->sharedFixture = NULL;
    }
}
```

Тесты для проекта магазина

Определимся со структурой. Создадим базовый абстрактный класс тестов, в котором будем вести подготовку к основному тесту.

```php
<?php
require_once "autoload.php";

abstract class BaseTest extends PHPUnit_Framework_TestCase{
    protected function setUp()
    {
        App::Init();
    }
}
```

Система должна знать правило загрузки классов, а также иметь рабочий экземпляр приложения.

Заметьте, что все **require** пишутся относительно корневой директории логики, а не директории **tests**.

Все последующие классы тестов будем наследовать от **BaseTest**. Создадим тест для класса категорий:

```
require_once "tests/BaseTest.php";
class CategoryTest extends BaseTest
{
    public function testGetCategories(){
        $this->assertNotNull(Category::getCategories());
    }
}
```

Подобным образом можем покрывать тестами любой из классов приложения.

К примеру, протестировать структуру в наборе данных контроллера:

```
class CategoriesControllerTest extends BaseTest
{
    /**
     * @dataProvider providerCategoriesController
     */
    public function testIndex($a){
        $cc = new CategoriesController();
        $cc_result = $cc->index($a);

        $this->assertNotNull($cc_result);
        $this->assertArrayHasKey("subcategories", $cc_result);
        $this->assertArrayHasKey("goods", $cc_result);
    }
    public function providerCategoriesController(){
        return array (
            array (["id" => 0]),
            array (["id" => 1]),
            array (["id" => 2])
        );
    }
}
```

Итоги

Мы познакомились с методиками тестирования кода и попробовали выполнить самые простые тесты на уже имеющемся коде. Важно понимать, что компании не любят переплачивать за то, что непосредственно не несет прибыль (а тесты никакой прибыли не несут – они избавляют от расходов).

Это везение, если заказчик не против оплачивать тестирование. Но всегда стоит попытаться донести до компании, что в долгосрочной перспективе тестирование сэкономит гораздо больше денег – за счет отсутствия инцидентов.

Практическое задание

1. Разобраться с принципом работы **PHPUnit**.
2. По образу модуля товаров из движка V1.0 создать модуль просмотра карточки товара:
    a. Карточка должна содержать в себе информацию о товаре, картинки, цену;
    b. Должна быть кнопка «Купить» (пока заглушка);
    c. Код должен быть покрыт тестами.
3. * Создать модуль просмотра результатов тестов.

Дополнительные материалы

1. https://habrahabr.ru/post/56289/ – юнит-тестирование в примерах.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Мэтт Зандстра. PHP. Объекты, шаблоны и методики программирования.